PHASE 1 DESIGN DOCUMENT

By: Shaul, Andy, Rachel, Yash, Elliot, Shridhar

Updated Specification: Our group is creating a Scrabble game that allows two or more human players to play against each other. Broadly, the Scrabble board is a 15 x 15 board of squares; some of the squares are equipped with certain special bonuses that will alter how a word that a player plays is calculated. When it is a player's turn, they can choose from one of three options: playing a word on the board, reshuffling their rack by putting some tiles back in the bag and redrawing, or passing their turn. If a word is placed on the board by a player, we check if it is a valid play by searching for the word in the Scrabble dictionary. A game of Scrabble ends when there are no letter tiles remaining in the bag, or the players have run out of tiles and valid moves to make. Then, the player with the highest total number of points after subtracting the points of the tiles remaining on their rack is the winner.

Since the Phase 0 deadline, we have worked on implementing different types of moves (swaps and passes; previously, we only had the move for placing tiles on the board), and checking whether placed word is valid (for instance, the first word on the board must go through the middle of the board, and any subsequent word must touch another tile on the board). We have also implemented a GUI and a GameState class for saving and loading a game.

Major Design Decisions: Originally for rendering the different components (board, tiles, squares, etc), we had the code structured so that each of these classes had its own render method. Upon further thought, we realized that structuring the code this way seemed to violate the Single Responsibility principle, since these classes were now additionally responsible for their own rendering. We also realized that this design would violate clean architecture as the rendering methods are part of the Interface Adapters layer, whereas the classes they were being implemented in were otherwise Entities. We decided that it would be better to create a Renderer class, responsible for all the rendering, and the code was refactored so that these rendering methods became part of the new Renderer class.

Clean Architecture: We have designed our classes, so that each falls under exactly one of the clean architecture layers. As explained above, this was still an issue until rendering was refactored. Our refactoring in general helped adhere more strongly to the clean architecture. By splitting some classes from phase 0 into several smaller classes we were able to more strongly demonstrate the division between the different layers. We also made sure to adhere to the clean architecture within our testing, by testing the business rules without relying on any external elements. We also maintained UI independence with the introduction of GUI.

SOLID Design Principles: We have tried to adhere to the SOLID principles as much as possible. For example, we have different classes for each different type of move (place, swap, passes) which adheres to the Single Responsibility Principle: each of these classes has only one reason

to change (that is, we alter a class when we want to modify that specific type of move). However, there are certain SOLID design principles that our code could benefit, one of which is the Interface Segregation Principle. Currently, we do not have many interfaces in our program; we plan on implementing more in Phase 2 as this will help not only with the SOLID design principles but also with Clean Architecture.

Packaging Strategies: As suggested in the lecture on packages, we mainly considered two packaging strategies: packaging by layer and packaging by component. Ultimately, we decided to use packaging by component, and decided to create one outer package encompassing everything, including the class containing our main method, and two additional packages within this package: one for the internal game data (ex. Tiles) and one for the GUI related functions. We used this packaging strategy since we thought it organized our classes in the cleanest manner and reflected encapsulation well. Furthermore, packaging by component allows classes that are coupled together to be in the same package, reducing the need for public methods and allowing us to make better use of package-private methods (although we still have to change many of our method access modifiers to make use of this advantage in our packaging strategy).

Design Patterns: *Command Design Pattern:* We used the Command design pattern to represent the different types of moves. The Command design pattern consists of an Order interface, which acts as a command, and concrete Command classes that implement the Orderinterface. It also consists of a Receiver, which places orders. In this case, the Move interface is the Order interface, and the PassMove, PlaceMove, and SwapMove classes correspond to the concrete command classes that inherit from the interface, and the main game loop places the orders for the moves. Our use of this design pattern is further highlighted by the use of the execute method signature in the Move interface.

Memento Design Pattern: We used the Memento design pattern for the GameState. The Memento design pattern consists of a Memento class, which contains the state of the object to be restored, the Originator class, which creates and stores Memento objects, and the CareTaker class, which restores object states from Memento. In this case, the GameState class is the Memento, the Game class is the Originator, and the ScrabbleGame class is the CareTaker. Indeed, we see that the methods in these classes are consistent with the design pattern; for example, the loadGameState and getGameState methods in Game correspond to the Originator loading and saving the Memento.

(Note: the following links were used to describe the design patterns: https://sourcemaking.com/design_patterns/command, https://www.tutorialspoint.com/design_pattern/memento_pattern.htm).

Progress Report: One aspect of our design that worked well was that because we aimed to adhere to clean architecture and worked on the inner layers before the outer layers, implementing outer layer classes such as the classes required for the GUI and the GameState was greatly facilitated.

Certain open questions we struggled with were how to save the GameState onto the disk. Currently, we are saving the GameState into three csv files; one representing the board (which letters are in what positions of the board, one representing the player information (the names, points, and racks of each player) and one representing the tiles left in the bag, the index of the current player in the list of players in PlayerManager, and the number of players in the game. This might not make the most sense, however; another option we will explore in Phase 2 is json files. We were also wondering where it would be most appropriate to save the files.

Finally, below is a summary of what everyone in the group has worked on so far, and what we will work on in the future:

Name	Work So Far	Future Work
Shaul	Checking whether a word is placed in a valid location, significant refactoring, much of the test suite.	Creating and implementing AI players of various difficulties.
Andy	Connecting GameState class to the game, packaging, significant refactoring	Implementing challenges (where another player contests whether a word placed on the board is valid)
Rachel	Implementing GameState class, checking that first word goes through the middle square	Writing GameState class tests, exploring other options of where/how to save the GameState
Yash	Checking whether a word is placed in a valid location, implementing other types of moves (ex. swaps)	Creating and implementing AI players of various difficulties.
Elliot	Creating a GUI using Swing. Methods to render Tiles, Squares, Board, Rack, Scoreboard. Rendering is now part of the main game loop.	Making the GUI interactive (currently, it only displays the rack, board and scoreboard; tiles must be placed in the console).
Shridhar	Implementing wild card tiles	Touching up the code so that it is ready to be merged into main and follows clean architecture